**Application of Greedy algorithm in operating system:**

Some examples are given below:

First Fit algorithm in Memory Management

Best Fit algorithm in Memory Management

Worst Fit algorithm in Memory Management

Shortest Job First Scheduling

**Where to use greedy Algorithm:**

1. If a problem has optimal substructure. The optimal solution to the problem contains optimal solution to the sub problem.

2.It has a greedy property. (hard to prove the correctness) If you make a choice that seems the best for the correct moment and solve the remaining sub-problems later, you will reach an optimal solution. That is why it is very hard to prove the correctness of it. Because, we are hoping if we choose the best solution for a sub problem without solving the other sub problem, we are hoping that it will somehow lead to the correct solution.

**Steps to follow for solving a problem using the greedy algorithm approach:**

1.Determine the optimal substructure of the problem/

2.Develop the recursive solution.

3.Show that if we make the greedy choice, only one sub-problem remains and that is the main problem. This is very important step.

4.Prove that it is always safe to make the greedy choice.

5.Convert the recursive solution into iterative one.

**How greedy choice is different from dynamic programming:**

In greedy choice property, we can assemble a globally optimized solution by making local optimal solutions. In other words, we are considering which choices to make, we make the choices that looks best for the current sub problem, without considering/worrying the result of sub-sub problems.

That is why in greedy algorithm paradigm, it is absolutely important to prove that there is only one sub-problem. (which is solution to main problem remains at last)

Now, in dynamic problem we make a choice at each step for current sub problem, but the result is depended on sub sub problems.

**Some Greedy Problems:**

**1. Activity Selection Problem:**

I can do it. However, we need to sort the list based on finish time first.

Now, how to sort it based on finish time.

The sort function in c++ algorithm header accepts one optional functional pointer. (an example of callback function right??)

Now, we can write a function like the following:

bool activityCompare(Activity s1,Activity s2)

{

return s1.finish<s2.finish;

}

And, call the sort function

sort(arr,arr+n,activityCompare);

**(An application of callback function)**

1. **Egyptian Fraction:**

Every positive fraction can be represented as sum of unique unit fractions. A fraction is unit fraction if numerator is 1 and denominator is a positive integer, for example 1/3 is a unit fraction. Such a representation is called Egyptian Fraction as it was used by ancient Egyptians.

Following are few examples:

Egyptian Fraction Representation of 2/3 is 1/2 + 1/6

Egyptian Fraction Representation of 6/14 is 1/3 + 1/11 + 1/231

Egyptian Fraction Representation of 12/13 is 1/2 + 1/3 + 1/12 + 1/156

**3.Job Sequencing Problem:**

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

**Examples:**

Input: Four Jobs with following deadlines and profits

JobID Deadline Profit

a 4 20

b 1 10

c 1 40

d 1 30

Output: Following is maximum profit sequence of jobs

c, a

Input: Five Jobs with following deadlines and profits

JobID Deadline Profit

a 2 100

b 1 19

c 2 27

d 1 25

e 3 15

Output: Following is maximum profit sequence of jobs

c, a, e

1) Sort all jobs in decreasing order of profit.

2) Initialize the result sequence as first job in sorted jobs.

3) Do following for remaining n-1 jobs

.......a) If the current job can fit in the current result sequence

without missing the deadline, add current job to the result.

Else ignore the current job.

**4.Huffman Coding:**

**Steps to build Huffman Tree:**

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

**Steps to print codes from Huffman Tree:**

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.

**Efficient Huffman Coding for Sorted Input:  
  
based on two queues.**

1. Create two empty queues.

2. Create a leaf node for each unique character and Enqueue it to the first queue in non-decreasing order of frequency. Initially second queue is empty.

3. Dequeue two nodes with the minimum frequency by examining the front of both queues. Repeat following steps two times

…..a) If second queue is empty, dequeue from first queue.

…..b) If first queue is empty, dequeue from second queue.

…..c) Else, compare the front of two queues and dequeue the minimum.

4. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first Dequeued node as its left child and the second Dequeued node as right child. **Enqueue this node to second queue.**

# Repeat steps#3 and #4 until there is more than one node in the queues. The remaining node is the root node and the tree is complete. Fractional Knapsack Problem:

Given weights and values of n items, we need put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In the [0-1 Knapsack problem](https://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/), we are not allowed to break items. We either take the whole item or don’t take it.

In Fractional Knapsack, we can break items for maximizing the total value of knapsack. This problem in which we can break item also called fractional knapsack problem.

Input :

Same as above

Output :

Maximum possible value = 240

By taking full items of 10 kg, 20 kg and

2/3rd of last item of 30 kg

A brute-force solution would be to try all possible subset with all different fraction but that will be too much time taking.

An efficient solution is to use **Greedy approach. The basic idea of greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with highest ratio and add them until we can’t add the next item as whole and at the end add the next item as much as we can. Which will always be optimal solution of this problem.**

A simple code with our own comparison function can be written as follows, please see sort function more closely, the third argument to sort function is our comparison function which sorts the item according to value/weight ratio in non-decreasing order.

After sorting we need to loop over these items and add them in our knapsack satisfying above mentioned criteria.

**// C/C++ program to solve fractional Knapsack Problem**

**#include <bits/stdc++.h>**

**using namespace std;**

**// Structure for Item which store weight and corresponding**

**// value of Item**

**struct Item**

**{**

**int value, weight;**

**// Constructor**

**Item(int value, int weight) : value(value), weight(weight)**

**{}**

**};**

**// Comparison function to sort Item according to val/weight ratio**

**bool cmp(struct Item a, struct Item b)**

**{**

**double r1 = (double)a.value / a.weight;**

**double r2 = (double)b.value / b.weight;**

**return r1 > r2;**

**}**

**// Main greedy function to solve problem**

**double fractionalKnapsack(int W, struct Item arr[], int n)**

**{**

**// sorting Item on basis of ration**

**sort(arr, arr + n, cmp);**

**// Uncomment to see new order of Items with their ratio**

**/\***

**for (int i = 0; i < n; i++)**

**{**

**cout << arr[i].value << " " << arr[i].weight << " : "**

**<< ((double)arr[i].value / arr[i].weight) << endl;**

**}**

**\*/**

**int curWeight = 0; // Current weight in knapsack**

**double finalvalue = 0.0; // Result (value in Knapsack)**

**// Looping through all Items**

**for (int i = 0; i < n; i++)**

**{**

**// If adding Item won't overflow, add it completely**

**if (curWeight + arr[i].weight <= W)**

**{**

**curWeight += arr[i].weight;**

**finalvalue += arr[i].value;**

**}**

**// If we can't add current Item, add fractional part of it**

**else**

**{**

**int remain = W - curWeight;**

**finalvalue += arr[i].value \* ((double) remain / arr[i].weight);**

**break;**

**}**

**}**

**// Returning final value**

**return finalvalue;**

**}**

**// driver program to test above function**

**int main()**

**{**

**int W = 50; // Weight of knapsack**

**Item arr[] = {{60, 10}, {100, 20}, {120, 30}};**

**int n = sizeof(arr) / sizeof(arr[0]);**

**cout << "Maximum value we can obtain = "**

**<< fractionalKnapsack(W, arr, n);**

**return 0;**

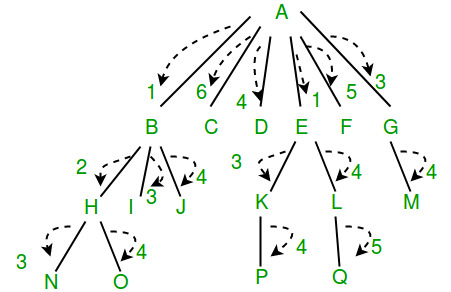
**}**

**Minimum Number Of Iterations To Pass Information To All Nodes In The Tree:**

Given a very large n-ary tree. Where the root node has some information which it wants to pass to all of its children down to the leaves with the constraint that it can only pass the information to one of its children at a time (take it as one iteration).

Now in the next iteration the child node can transfer that information to only one of its children and at the same time instance the child’s parent i.e. root can pass the info to one of its remaining children. Continuing in this way we have to find the minimum no of iterations required to pass the information to all nodes in the tree.

Minimum no of iterations for tree below is 6. The root A first passes information to B. In next iteration, A passes information to E and B passes information to H and so on.



This can be done using Post Order Traversal. The idea is to consider height and children count on each and every node.

If a child node i takes ci iterations to pass info below its subtree, then its parent will take (ci + 1) iterations to pass info to subtree rooted at that child i.

If parent has more children, it will pass info to them in subsequent iterations. Let’s say children of a parent takes c1, c2, c3, c4, …, cn iterations to pass info in their own subtree, Now parent has to pass info to these n children one by one in n iterations. If parent picks child i in ith iteration, then parent will take (i + ci) iterations to pass info to child i and all it’s subtree.

In any iteration, when parent passes info a child i+1, children (1 to i) which got info from parent already in previous iterations, will pass info to further down in subsequent iterations, if any child (1 to i) has its own child further down.

Nodes with height = 0: (Trivial case) Leaf node has no children (no information passing needed), so no of iterations would be ZERO.

Nodes with height = 1: Here node has to pass info to all the children one by one (all children are leaf node, so no more information passing further down). Since all children are leaf, node can pass info to any child in any order (pick any child who didn’t receive the info yet). One iteration needed for each child and so no of iterations would be no of children.So node with height 1 with n children will take n iterations.

Take a counter initialized with ZERO, loop through all children and keep incrementing counter.

Nodes with height > 1: Let’s assume that there are n children (1 to n) of a node and minimum no iterations for all n children are c1, c2, …., cn.

To make sure maximum no of nodes participate in info passing in any iteration, parent should 1st pass info to that child who will take maximum iteration to pass info further down in subsequent iterations. i.e. in any iteration, parent should choose the child who takes maximum iteration later on. It can be thought of as a greedy approach where parent choose that child 1st, who needs maximum no of iterations so that all subsequent iterations can be utilized efficiently.

If parent goes in any other fashion, then in the end, there could be some nodes which are done quite early, sitting idle and so bandwidth is not utilized efficiently in further iterations.

If there are two children i and j with minimum iterations ci and cj where ci > cj, then If parent picks child j 1st then no of iterations needed by parent to pass info to both children and their subtree would be:max (1 + cj, 2 + ci) = 2 + ci

If parent picks child i 1st then no of iterations needed by parent to pass info to both children and their subtree would be: max(1 + ci, 2 + cj) = 1 + ci (So picking ci gives better result than picking cj)

So,

sort all ci values decreasing order,  
let’s say after sorting, values are c1 > c2 > c3 > …. > cn  
**take a counter c, set c = 1 + c1 (for child with maximum no of iterations)  
for all children i from 2 to n, c = c + 1 + ci**

**(c is a counter value)**

C is the counter value. Starting from 0 and incremented by 1 after every step

Ci is the needed number of iterations for ith node to pass the information to it’s subtree.